TESINA ALGORITMI E STRUTTURE DATI

Codici di Huffman

Luca Tombesi
Simone Pesci

SOMMARIO

PROBLEMA AFFRONTATO

In questa sezione del documento ci si occupa di dare una spiegazione del problema affrontato.

Nella teoria dell'informazione, per **codifica di Huffman** si intende un algoritmo di codifica dei simboli usato per la compressione di dati, basato sul principio di trovare il sistema ottimale per codificare stringhe a seconda della frequenza relativa di ciascun carattere. Dato un testo da comprimere, si crea una tabella per la memorizzazione delle frequenze di ciascun carattere nel testo. Per l'algoritmo ci si basa sull'idea che caratteri che si presentano con maggiore frequenza verranno codificati usando meno bit mentre per caratteri che si presentano più raramente si useranno più bit. Come esempio , si supponga di avere un file di 100000 caratteri e si supponga che il testo contenga soltanto 6 caratteri con le frequenze di seguito indicate (in migliaia):

carattere	a	b	c	d	e	f
frequenza	45	13	12	16	9	5

Se utilizziamo un codice a lunghezza fissa (ad esempio quello mostrato sotto) abbiamo bisogno di tre bit per ogni carattere per un numero di bit complessivo per l'intero file pari a 300000.

carattere	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
codice	000	001	010	011	100	101

Figura 1: codifica a lunghezza fissa

Se invece si utilizza un codice a lunghezza variabile (come quello in Figura 2 di seguito), il numero di bit complessivi è pari a $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224000$.

carattere	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
codice	0	101	100	111	1101	1100

Figura 2: codifica a lunghezza variabile

Il codice a lunghezza variabile (2) è un codice prefisso, cioè un codice in cui nessuna parola di codice è prefisso di un'altra parola di codice. I codici prefissi semplificano la decodifica, poiché le parole di codice possono essere decodificate senza ambiguità. Il codice prefisso è rappresentato mediante un albero binario, che può essere utilizzato nella fase di decodifica per individuare facilmente le parole di codice. L'albero binario associato ad un codice prefisso è definito come segue:

- ogni foglia dell'albero è un carattere;

- ogni arco padre-figlio è associato al valore 0 o 1 di un bit (figlio sinistro=0, figlio destro=1);

- la parola di codice associata ad un carattere è la sequenza di bit lungo il cammino dalla radice alla foglia;

Nel seguito etichetteremo le foglie con il carattere associato e la sua frequenza, e i nodi interni con la somma delle frequenze dei caratteri nel sottoalbero. Un codice ottimo (che porta cioè ad una compressione ottima dei dati) corrisponde sempre ad un albero binario in cui ogni nodo interno ha esattamente due figli; se indichiamo con f(c) la frequenza del carattere c e con d_T (c) la profondità della foglia che rappresenta c in T , il costo dell'albero (cioè il numero di bit necessari a codificare il testo) è

$$B(T) = \sum_{c \in C} d_T(c) f(c) ,$$

dove C è l'insieme di tutti i caratteri. Lo scopo di questo progetto è la creazione di un programma che prende in input un insieme di caratteri, ne calcola la tabella delle frequenze, calcola il codice di Huffman e permette di codificare e decodificare il testo. Inoltre, la misurazione del fattore di compressione e dei tempi di esecuzione delle varie operazioni deve permettere di verificare l'efficienza dell'algoritmo, che risulterà massima nel caso in cui si provveda ad implementare un codice prefisso ottimo.

COSTRUZIONE CODICE PREFISSO OTTIMO

Il problema che vogliamo affrontare è il seguente: dato un alfabeto di caratteri C e una frequenza f(c) per ogni carattere c ∈ C, vogliamo creare un codice prefisso ottimo per C in base alle frequenze date. Poiché un codice prefisso ottimo può essere descritto in maniera univoca mediante un albero binario, il problema può essere descritto anche come segue: dato un alfabeto di caratteri C e una frequenza f(c) per ogni carattere c ∈ C, vogliamo costruire un albero corrispondente ad un codice prefisso ottimo.

Definizione sottoproblemi

Dato il problema definito da C , possiamo ricondurre la soluzione di tale problema alla risoluzione di un sottoproblema come segue. Supponiamo di saper determinare un codice prefisso ottimo (e quindi il corrispondente albero T') per un sottoinsieme C′ dei caratteri di C. Definiamo un carattere fittizio c* la cui frequenza è data da

$$f(c^*) = \sum_{c \in C'} f(c)$$

Consideriamo adesso l'alfabeto C″ = C ∖ C′ ∪ {c*} con le corrispondenti frequenze, e calcoliamo un codice prefisso ottimo per C″(e quindi il corrispondente albero T″). In T″ , il carattere c* compare come una foglia (in termini di codice significa che al carattere c* è associata una certa parola di codice); se rimpiazziamo la foglia che rappresenta c* con l'albero T' otteniamo un nuovo albero T‴ le cui foglie sono i caratteri di C. In termini di codice, la trasformazione di T″ in T‴ significa che la parola di codice di ciascun carattere c ∈ C′ è data dalla concatenazione della parola di codice che rappresenta c* in T'' e della parola di codice che rappresenta c in T′ . Per quanto riguarda la scelta del sottoinsieme C′ e il calcolo del corrispondente albero T′ ,osserviamo che se C′ è costituito da due caratteri, il corrispondente T′ è banale, essendo un albero con una radice e due figli corrispondenti ai due caratteri scelti,uno codificato come 0 e l'altro come 1. In definitiva, un codice prefisso può essere calcolato scegliendo due caratteri, costruendo un albero di codifica banale, sostituendo i due caratteri con un carattere fittizio la cui frequenza è pari alla somma di quelli scelti e procedendo allo stesso modo sul nuovo insieme di caratteri.

Sottostruttura ottima

Dimostriamo che il problema di costruire un codice prefisso ottimo gode della proprietà della sottostruttura ottima.

Teorema 1 *Sia C un alfabeto dove ogni carattere c ∈ C ha frequenza f(c) e sia T un albero che rappresenta un codice prefisso ottimo per C . Siano x e y due caratteri in C la cui codifica differisce solo per l'ultimo bit (in altri termini x e y hanno lo stesso padre in T). Sia C' l'alfabeto definito come C' = C ∖ {x, y} ∪ {z} , in cui z è un carattere fittizio per il quale definiamo la frequenza f(z) = f(x) + f(y) . Sia T' l'albero che si ottiene da T sostituendo il sottoalbero radicato nel padre delle foglie che rappresentano x e y con un nodo foglia che rappresenta z . T' rappresenta un codice prefisso ottimo per C' .*

Prova:

Il costo di T è:

$$B(T) = \sum_{c \in C} d_T(c) f(c)$$

mentre il costo di T' è:

$$B(T') = \sum_{c \in C'} d_{T'}(c) f(c)$$

Abbiamo $d_T(c)f(c) = d_{T'}(c)f(c)$ per ogni $c \in C \setminus \{x, y\}$; d'altra parte poichè $d_T(x) = d_T(y) = d_{T'}(z)+1$, abbiamo

$$d_T(x)f(x) + d_T(y)f(y) = \left(d_{T'}(z) + 1\right)(f(x) + f(y)) = d_{T'}(z)f(z) + f(x) + f(y)$$

Si ottiene quindi:

$$B(T) = \sum_{c \in C} d_T(c) f(c)$$

$$= \sum_{c \in C' \setminus \{x,y\}} d_T(c) f(c) + d_T(x) f(x) + d_T(y) f(y)$$

$$= \sum_{c \in C' \setminus \{x,y\}} d_{T'}(c) f(c) + d_{T'}(z) f(z) + f(x) + f(y)$$

$$= B(T') + f(x) + f(y)$$

Supponiamo per assurdo che T' rappresenti un codice prefisso per C' non ottimo. Esisterà allora un albero T'' che corrisponde ad un codice prefisso per C' e tale che B(T'') < B(T'). Poichè $z \in C'$, esisterà una foglia di T'' che rappresenta z. Sostituiamo z in T'' con un nodo interno che ha per figli due nodi che rappresentano x e y. L'albero T''' così ottenuto rappresenta un codice prefisso per C il cui costo è: B(T''') = B(T'') + f(x) + f(y). Dall'ipotesi che B(T'') < B(T') abbiamo:

$$B(T''') = B(T'') + f(x) + f(y) < B(T') + f(x) + f(y) = B(T)$$

ma ciò contraddice l'ipotesi che T sia ottimo.

Scelta golosa

Come abbiamo detto per costruire un codice prefisso ottimo possiamo ad ogni passo scegliere due caratteri per i quali calcolare una soluzione banale e poi affrontare un nuovo problema in cui i due caratteri sono sostituiti da un carattere fittizio. Adottiamo come scelta golosa quella che consiste nello scegliere i due caratteri con frequenza più bassa. Dimostriamo che con tale scelta il problema gode della proprietà della scelta golosa.

Teorema 2 *Sia C un alfabeto dove ogni carattere $c \in C$ ha frequenza f(c). Siano x e y due caratteri in C con le frequenze più basse. Esiste un codice prefisso ottimo per C*

rappresentato da un albero in cui le foglie che rappresentano x e y hanno lo stesso padre e si trovano a profondità massima.

Prova: Sia T un albero che rappresenta un codice prefisso ottimo per C. Se x e y sono rappresentati in T da due foglie sorelle a profondità massima, allora T è l'albero cercato e la prova termina. Supponiamo quindi che le foglie che rappresentano x e y in T non siano sorelle a profondità massima (cioè o non sono sorelle o non sono a profondità massima). Trasformiamo T in un nuovo albero T' in cui x e y sono foglie sorelle a profondità massima e tale che B(T') = B(T) (cioè T' è ancora una soluzione ottima). Siano b e d due foglie sorelle di profondità massima in T. Senza perdita di generalità assumiamo f(x) ≤ f(y) e f(b) ≤ f(d). Poiché x e y sono i due caratteri a frequenza minima, abbiamo f(x) ≤ f(b) e f(y) ≤ f(d).
Sia T'' l'albero che si ottiene scambiando x con b e sia T' l'albero che si ottiene scambiando y con d. Ovviamente le foglie che rappresentano x e y in T' sono sorelle. Dobbiamo dimostrare che B(T') = B(T). Per farlo dimostriamo che B(T'') = B(T) e che B(T') = B(T'').
Tenendo presente che $d_{T''}(x) = d_T(b)$ e $d_{T''}(b) = d_T(x)$, abbiamo:

$$B(T) - B(T'') = \sum_{c \in C} f(c)\, d_T(c) - \sum_{c \in C} f(c)\, d_{T''}(c)$$

$$= f(x)\, d_T(x) + f(b)\, d_T(b) - f(x)\, d_{T''}(x) - f(b)\, d_{T''}(b)$$

$$= f(x)\, d_T(x) + f(b)\, d_T(b) - f(x)\, d_T(b) - f(b)\, d_T(x)$$

$$= (f(b) - f(x))\left(d_T(b) - d_T(x)\right)$$

Poiché f(x) ≤ f(b) abbiamo (f(b) − f(x)) ≥ 0, e poiché $d_T(b) \geq d_T(x)$ abbiamo $(d_T(b) - d_T(x)) \geq 0$; ne segue che B(T) ≥ B(T''). Poiché T è ottimo non può essere B(T) > B(T'') e quindi B(T) = B(T''). Procedendo in maniera analoga otterremo che B(T'') = B(T').
In che senso la scelta descritta è una scelta golosa? Si può dimostrare (vedi Lemma 1) che il costo di un albero T può essere calcolato come

$$B(T) = \sum_{v \in int(T)} f(v)$$

in cui *int(T)* denota l'insieme dei nodi interni di *T* e *f(v)* indica la frequenza associata a v, cioè la somma delle frequenze dei caratteri le cui foglie appartengono al sottoalbero radicato in v. Quando scegliamo due caratteri x e y e calcoliamo un albero che li codifica, l'albero creato ha un costo che è uguale a f(x) + f(y) (tale infatti è la frequenza associata all'unico nodo interno dell'albero creato). Pertanto la scelta di selezionare x e y come i caratteri di minor frequenza, porta alla creazione di un albero di minimo costo tra tutti quelli che si possono creare scegliendo due caratteri.

Lemma 1 *Sia T un albero che rappresenta un codice prefisso. Abbiamo*

$$B(T) = \sum_{c \in C} d_T(c) f(c) = \sum_{v \in int(T)} f(v)$$

Prova: La prova è per induzione sul numero n di nodi interni.

n = 1. Sia u l'unico nodo interno (cio`e il padre dei due nodi che rappresentano x e y). Abbiamo:

$$\sum_{c \in C} d_T(c) f(c) = f(x) + f(y) \quad e \quad \sum_{v \in int(T)} f(v) = f(u) = f(x) + f(y)$$

n > 1. Sia u la radice di T, siano u1 ed u2 i figli di u e siano T1 e T2 i sottoalberi di T radicati in u1 e u2 rispettivamente. Indichiamo inoltre con C_i il sottoinsiemi di C che contiene i caratteri rappresentati da foglie di T_i (i = 1, 2).
Abbiamo:

$$B(T) = \sum_{c \in C} d_T(c) f(c)$$

$$= \sum_{c \in C_1} \left(d_{T_1}(c) + 1 \right) f(c) + \sum_{c \in C_2} \left(d_{T_2}(c) + 1 \right) f(c)$$

$$= \sum_{c \in C_1} d_{T_1}(c) f(c) + \sum_{c \in C_1} f(c) + \sum_{c \in C_2} d_{T_2}(c) f(c) + \sum_{c \in C_2} f(c)$$

$$= B(T_1) + B(T_2) + \sum_{c \in C_1} f(c) + \sum_{c \in C_2} f(c)$$

Per induzione:

$$B(T_1) = \sum_{v \in int(T_1)} f(v) \qquad e \qquad B(T_2) = \sum_{v \in int(T_2)} f(v)$$

Inoltre per definizione si ha:

$$f(u_1) = \sum_{c \in C_1} f(c) \qquad e \qquad f(u_2) = \sum_{c \in C_2} f(c)$$

Ne segue:

$$B(T) = \sum_{v \in int(T_1)} f(v) + \sum_{v \in int(T_2)} f(v) + \left(f(u_1) + f(u_2) \right)$$

$$= \sum_{v \in int(T_1)} f(v) + \sum_{v \in int(T_2)} f(v) + f(u)$$

$$= \sum_{v \in int(T)} f(v)$$

Algoritmo greedy per la costruzione di un codice prefisso ottimo

Di seguito mostriamo lo pseudo-codice di un algoritmo greedy per il calcolo di un codice prefisso ottimo.

```
HUFFMAN(C, f )
    n = |C|
    Q = nuova coda di priorità vuota
    for each c ∈ C
        alloca un nuovo nodo c
        c.freq = f(c) //il campo freq è usato come chiave nella coda di priorità
        INSERT(Q,c)
    for i = 1 to n − 1
        alloca un nuovo nodo z
        z.left = x = EXTRACT-MIN(Q)
        z.right = y = EXTRACT-MIN(Q)
        z.freq = x.freq + y.freq
        INSERT(Q,z)
    return EXTRACT-MIN(Q) //Restituisce la radice dell'albero
```

Lo pseudocodice prende un insieme di caratteri con le loro frequenze, crea una coda vuota di minima priorità e per ogni carattere crea il corrispettivo nodo a cui viene associato all'attributo *freq* la frequenza di tale carattere nel testo, quindi lo inserisce nella coda. Il campo freq del nodo viene usato come chiave nella coda *Q*.

Per n-1 volte viene poi allocato un nuovo nodo *z* prendendo i due nodi con frequenza minima nella coda che vengono assegnati come figlio sinistro e figlio destro di *z*. Al nodo creato si assegna come frequenza la somma delle frequenze dei nodi estratti dalla coda. Quindi si inserisce nella coda il nodo *z*.

L'algoritmo termina estraendo l'unico nodo rimasto nella coda ovvero la radice dell'albero di Huffman.

IMPLEMENTAZIONE STRUTTURE DATI

In questo codice vengono implementate le seguenti strutture dati:

- **CODA DI MIN-PRIORITÀ:** la coda è stata implementata tramite la classe “**minHeap**” utilizzando una rappresentazione tramite array. In particolare è stato creato un array di elementi di tipo *BinaryTreeNode*, dove la classe *BinaryTreeNode* rappresenta un nodo dell’albero binario. All’interno della coda, la priorità è determinata dalla frequenza dei nodi stessi ed essendo questa una coda di minima priorità il nodo con frequenza minore occuperà la prima posizione dell’array.

- **ALBERO BINARIO:** l’albero binario è stato implementato tramite la classe “**BinaryTreeNode**”. In particolare si è partiti dalla coda di minima priorità contenente i nodi con le varie frequenze e tramite gli attributi *getLeft(), getRight(), setLeft(BinaryTreeNode n) e setRight(BinaryTreeNode n)* sono stati aggiunti gli altri nodi fino ad arrivare ad avere un albero binario che rispetta le proprietà dell’albero di Huffman.

IMPLEMENTAZIONE DEGLI ALGORITMI E ANALISI DI COMPLESSITÀ

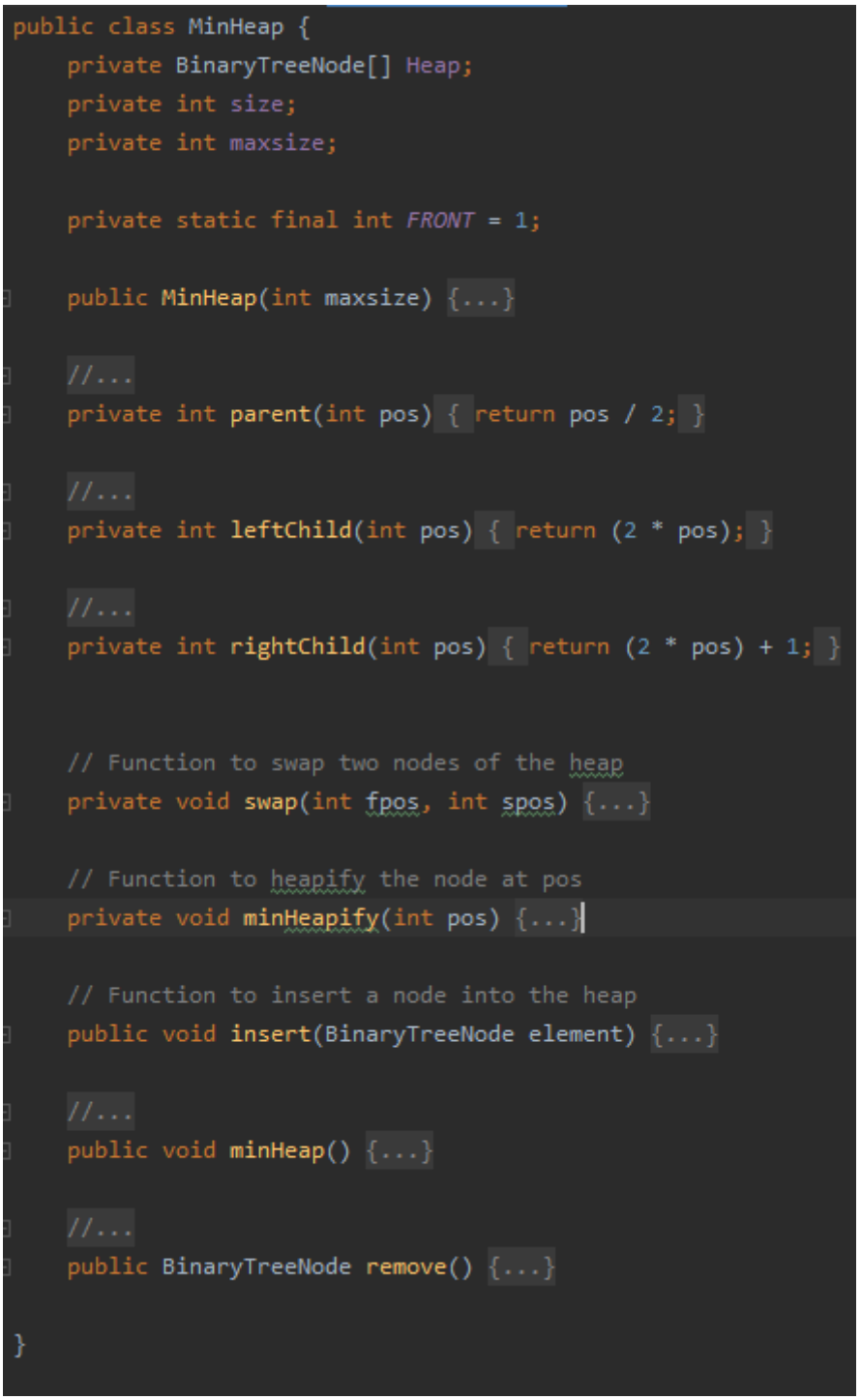
In questa sezione si illustrano le scelte implementative effettuate, discutendone anche la complessità mediante una descrizione delle classi utilizzate

```
public static BinaryTreeNode[] createTab() throws IOException {
    long l1 = System.nanoTime();
    BinaryTreeNode[] n=new BinaryTreeNode[10501];   // Length of the array which allows to code Russian, Greek characters and also to code ASCII arts
    int k=0;
    String f = "Resources\\prova.txt";
    Path l = Paths.get(f);
    content = Files.readString(l , StandardCharsets.UTF_8);


    for(char c=(char)(0);c<=(char)(10500);c++){
        n[(int)(c)]=new BinaryTreeNode(c, freq: 0);
        if( (int)(c) > max)
            max = c;
    }

    for(int i=0;i<content.length();i++)
        if(n[content.charAt(i)].character==content.charAt(i))
            n[content.charAt(i)].freq++;
    long l2 = System.nanoTime();
    System.out.println("tempo lettura:"+(l2-l1)/1E6);
    return n;
}
```

Il metodo **createTab()** crea un array di elementi BinaryTreeNode contenenti per ogni carattere, la frequenza con cui sono presenti nel testo da codificare “prova.txt”, i cui nodi verranno poi inseriti nel minHeap per l’algoritmo di Huffman.



```java
public class MinHeap {
    private BinaryTreeNode[] Heap;
    private int size;
    private int maxsize;

    private static final int FRONT = 1;

    public MinHeap(int maxsize) {...}

    //...
    private int parent(int pos) { return pos / 2; }

    //...
    private int leftChild(int pos) { return (2 * pos); }

    //...
    private int rightChild(int pos) { return (2 * pos) + 1; }


    // Function to swap two nodes of the heap
    private void swap(int fpos, int spos) {...}

    // Function to heapify the node at pos
    private void minHeapify(int pos) {...}

    // Function to insert a node into the heap
    public void insert(BinaryTreeNode element) {...}

    //...
    public void minHeap() {...}

    //...
    public BinaryTreeNode remove() {...}

}
```

La classe minHeap è necessaria per creare la coda di minima priorità.
Si è deciso di far partire l'heap dalla posizione *1* mettendo in posizione *0* un elemento con frequenza *Integer.MIN_VALUE* in modo tale da seguire l'implementazione fatta a lezione e rendere più semplice la gestione dei figli.

```
//...
private int parent(int pos) {
    return pos / 2;
}

//...
private int leftChild(int pos) {
    return (2 * pos);
}

//...
private int rightChild(int pos) {
    return (2 * pos) + 1;
}
```

I metodi **parent**(int pos) , **leftChild**(int pos) e **rightChild**(int pos) sono i metodi per la navigazione dei nodi dello heap. Il nodo i-esimo avrà padre i/2 e figli 2*i (il sinistro) e 2*i +1 (il destro).

```
// Function to swap two nodes of the heap
private void swap(int fpos, int spos) {
    BinaryTreeNode tmp;
    tmp = Heap[fpos];
    Heap[fpos] = Heap[spos];
    Heap[spos] = tmp;
}
```

Il metodo **swap**(int fpos, int spos) ci permette di scambiare due elementi dell'heap dati gli indici degli elementi da scambiare.

```
private void minHeapify(int pos) {


  int min=pos;
  if(leftChild(pos)<=size && Heap[min].freq > Heap[leftChild(pos)].freq)
      min=leftChild(pos);

      if(rightChild(min)<=size && Heap[min].freq > Heap[rightChild(pos)].freq)
          min=rightChild(pos);

      if(min!=pos ){
          swap(pos, min);
          minHeapify(pos);
      }
  }
```

Il metodo **minHeapify**(int pos) è fondamentale per la coda e ci permette (partendo dai due sottoalberi radicati in *leftChild*(pos) e in *rightChild*(pos) già minHeap) di ottenere un minHeap dell'albero di radice *pos*.
La complessità della procedura minHeapify(pos) è descritta dalla ricorrenza:

$$T(n) \leq T\left(\left(\frac{2}{3}\right) \cdot n\right) + \Theta(1)$$

Da cui abbiamo: $T(n) = O(\lg n)$

```
// Function to insert a node into the heap
public void insert(BinaryTreeNode element) {
    if (size >= maxsize) {
        return;
    }
    Heap[++size] = element;
    int current = size;

    while (current>1 && Heap[parent(current)].freq>Heap[current].freq) {
        swap(current, parent(current));
        current = parent(current);
    }
}
```

La procedura **insert**(BinaryTreeNode element) permette di inserire un nodo nell'heap: viene incrementata la grandezza dell'array (*size*) e viene aggiunto l'elemento in ultima posizione. Se poi tale elemento ha frequenza minore o uguale a quella del padre allora tramite il metodo *swap* facciamo salire il nodo nell'albero per far ritornare l'albero un minHeap. Nel caso peggiore abbiamo che si risale tutto l'albero, quindi avremo una complessità $O(\lg n)$.

```
// Function to build the min heap using
// the minHeapify
public void minHeap() {
    for (int pos = (size / 2); pos >= 1; pos--) {
        minHeapify(pos);
    }
}
```

Il metodo *minHeap*() ci permette di ripristinare le proprietà del minHeap.

```java
// Function to remove and return the minimum
// element from the heap
public BinaryTreeNode remove() {
    BinaryTreeNode popped = Heap[FRONT];
    Heap[FRONT] = Heap[size--];
    minHeapify(FRONT);
    return popped;
}
```

Il metodo **remove**() rimuove l'elemento di frequenza minore dell'albero, in particolare viene spostato l'elemento di posizione *FRONT* (ovvero in prima posizione) al posto di quello in ultima posizione, successivamente la dimensione dell' heap (*size*) viene decrementata e viene eseguita la procedura minHeapify(FRONT) ripristinando le proprietà dello heap. Anche qui nel caso peggiore si risale tutto l'albero, quindi avremo una complessità $O(\lg n)$.

```java
public static BinaryTreeNode HuffmanTree(BinaryTreeNode BTN[]){
    int count = 0 ;

    for (int i = 0; i < BTN.length ; i++){

        if(BTN[i].freq != 0)
            count++;

    }

    MinHeap Q = new MinHeap(count);

    for(int i=0; i < BTN.length   ; i++){

        if(BTN[i].freq != 0)
            Q.insert(BTN[i]);

    }

    for (int i = 0 ; i < count - 1  ; i++){

        Q.minHeap();

        BinaryTreeNode newNode = new BinaryTreeNode((char)(0), freq: 0);

        newNode.setLeft(Q.remove());
        newNode.setRight(Q.remove());


        newNode.freq = newNode.getLeft().freq + newNode.getRight().freq;
        Q.insert(newNode);

    }
    return Q.remove();

}
```

La procedura **HuffmanTree**(BinaryTreeNode BTN[]) ci permette di creare un albero di Huffman. In particolare, si implementa lo pseudocodice dell'algoritmo goloso relativo alla costruzione di un codice prefisso ottimo. Le operazioni relative al minHeap hanno una complessità di caso peggiore pari a $O(lg\ n)$, quindi , nel caso peggiore della creazione dell'albero impiegheremo un tempo $O(nlg\ n)$.

```java
public static String codifica(BinaryTreeNode n,String s) throws IOException {
    long c1 = System.nanoTime();
    StringBuilder codifica= new StringBuilder();
    String [] codArray=new String[max+1];
    search_ric(n,codArray, string: "");
    for(int i=0;i<s.length();i++){
        codifica.append(codArray[(int)(s.charAt(i))]);
    }


    long c2 = System.nanoTime();
    System.out.println("tempo codifica:"+(c2-c1)/1E6);
    return codifica.toString();
}

private static void search_ric(BinaryTreeNode n,String[] s,String string){
    if(n.getRight()==null && n.getLeft()==null) {
        s[n.character]=string;
    }
    else{

        if(n.getLeft()!=null) {
            search_ric(n.getLeft(),s, string: string+"0");
        }
        if(n.getRight()!=null) {
            search_ric(n.getRight(),s, string: string+"1");
        }

    }
}
```

Per la procedura di codifica utilizziamo i metodi **codifica**(BinaryTreeNode n , String s) e **search_ric**(BinaryTreeNode n, String[] s,String string).Viene creato un array di stringhe che memorizza la codifica di ogni carattere attraverso il metodo ***search_ric*** .Quindi si codifica l'intero testo attraverso il ciclo *for*. La complessità complessiva è $O(nlg\ n)$.

```java
public static String decodifica(BinaryTreeNode n,String s) {
    long d1 = System.nanoTime();
    StringBuilder decodifica = new StringBuilder();
    BinaryTreeNode n1 = n;
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == '0') {
            n1 = n1.getLeft();

        }
        else {
            n1 = n1.getRight();

        }
        if (n1.getLeft() == null && n1.getRight() == null) {
            decodifica.append(n1.character);
            n1 = n;
        }
    }

    long d2 = System.nanoTime();

    System.out.println("tempo decodifica : " + (d2-d1)/1E6);

    return decodifica.toString();
}
```

Il metodo **decodifica**(BinaryTreeNode n , String s) permette di ottenere la decodifica del testo codificato, anche qui si avrà una complessità di $O(nlg\ n)$.

ESEMPIO APPLICATIVO

Di seguito viene mostrato l'output del programma basandosi sul testo "prova.txt" (i tempi di esecuzione potrebbe variare da un'esecuzione ad un'altra).

Codice di Huffman calcolato: (0,9 ms)

E-->00000000

h-->00000001

D-->00000010

U-->00000011

q-->000001

p-->00001

l-->0001

i-->001

r-->0100

n-->0101

f-->0110000

x-->0110001

v-->0110010

g-->0110011

L-->01101000

"-->01101001

b-->0110101

.-->0110110

,-->0110111

u-->0111

a-->1000

t-->1001

 -->101

o-->1100

c-->11010

m-->11011

d-->11100

s-->11101

e-->1111

5,8 ms → tempo impiegato per codificare il testo con Huffman

01101001011010001100010011111101110100100001111010111110111011110011000001110
00100101111010011001101100011011111110010110111101110101100010111101111111010
1001111110010111010010110001110000100001001111011101000101010110011101111100
0100110010110111101111011111111001011110011001011111001011111101110111100111 00
1011001111111011000011100010010100101011101000111100001111000111010110011010
1111001101000110000110101110001001111101111110011011110011000001110001001111 1
0111011100001100110101100010110000000100100000101111000011011010100000011100
11011111010100111011101100011100101110110010101001110111010110010111101010011
00011011011011110100000101110011110110101011100111011001010001111110010111110
1100011111010011010001100110001001001110001011010111000100011000110111101011
00101000110000110101110001000011110110101010011110100110101111001101100000 01
00100000010111001000011011111011000110111111000101110101100110111101111001110 0
1100101110101100010111101111100000101111000100101101101010000000100111001111 0
1101100001111001111110100101000111010011111011110011000001110001001010010101
1010100111100001010011110000000111110101111001111010000110011010010101101011
0010110000001011100001100110001001111110101100101111000100110011011111111101111
0111111011101000100010001011111011101111001100000111000100111101111110111101 0
1100000111011001100110001001101010101110001000110001010000110000100001100 01
0010111010001101101010000000000110001110101111000011001111101110100101111010 0
1010110011011100110101101010001111110101000100110111010011100001001111001000
1001100010011010101110001011010000101001100001111001111010110010110111101111
0101110101100110100101011011101001110001000011000101000001011100110111000 11
00000110000000111010001100010111100111111101111101000111010110011011101111000 0

01000100110011011000010100111011101001111001011111111011001101000110000110101111000100011111011011011001101001

0,31 ms→ tempo impiegato per tradurre il testo codificato in testo originale

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

DATI SPERIMENTALI

In questa sezione dopo aver analizzato e il codice e le sue complessità si passa all'analisi dei dati sperimentali. I parametri di interesse sono in particolare il numero di caratteri del testo impiegato, lunghezza del testo con codifica a 8 bit e lunghezza del testo con codifica di Huffman.

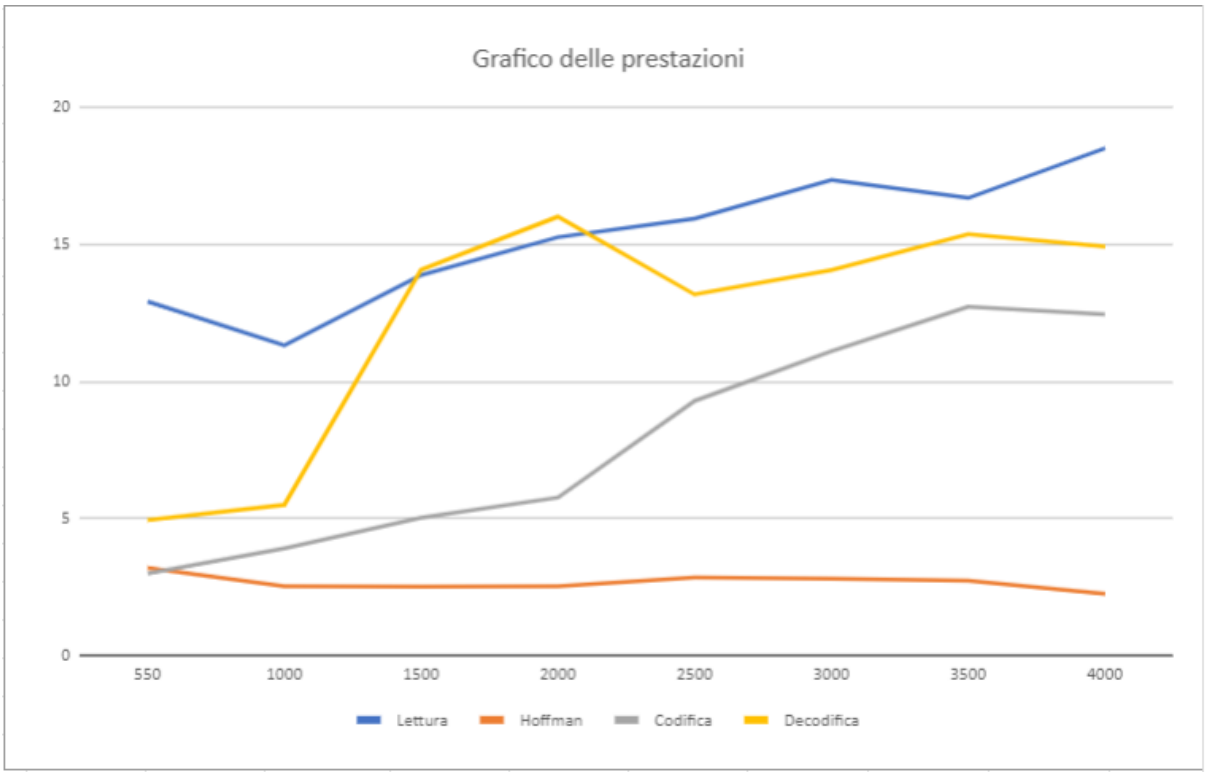
Numero di parole del testo preso in esame: 4000 parole.
Numero di bit con codifica 8 bit: 27941 caratteri *8=219928 bit.
Numero di bit del testo con codifica di Huffman:114013 bit.

Nella tabella seguente vengono mostrati i tempi in millisecondi delle varie sezioni del codice per ogni esecuzione:

Numero esecuzione	1	2	3	4	5	6	7	8	9	10
Tempo lettura testo	16,3	15,8	18,9	16,6	16,6	24,3	17,1	18,9	24,0	16,2
Creazione albero Huffman	2,5	2,1	2,4	1,9	2,4	2,5	2,1	2,3	2,5	2,2
Tempo di codifica	14,3	12,4	9,3	12,2	15,0	13,8	11,2	12,3	10,4	13,4
Tempo di decodifica	28,4	17,2	25,3	27,6	22,8	24,3	16,5	16,2	20,6	20,6

In seguito sono state effettuate diverse prove su testi casuali con un numero di parole crescenti, misurando i tempi di lettura , di creazione dell'albero di Huffman,di codifica del testo e di decodifica in modo tale da valutare l'efficienza e la correttezza dell'algoritmo implementato all'aumentare del numero di caratteri preso in considerazione:

Numero parole	Caratteri testo	Lettura	Huffman	Codifica	Decodifica
550	3775	12,9	3,2	2,9	4,9
1000	6867	11,3	2,5	3,9	5,4
1500	10300	13,8	2,5	5,0	14,0
2000	13734	15,2	2,5	5,8	16,0
2500	17058	15,9	2,6	16	18,9
3000	20487	17,3	2,5	12	21,4
3500	23970	16,6	2,7	14	23,7
4000	27331	18,5	2,4	16	21,9



Nella seguente tabella vengono invece mostrati i dati sperimentali riguardanti il fattore di compressione (ovvero il rapporto tra il numero di bit del testo originale e il numero di bit del testo codificato). Dalla tabella notiamo come la codifica di Huffman ci permetta di ottenere un risparmio notevole in termini di bit riducendo il numero di caratteri di un fattore di quasi 2.

Numero parole	Caratteri testo	Bit testo originale	Bit dopo la codifica	Fattore di compressione
550	3775	30200	15650	1,929
1000	6867	54936	28481	1,928
1500	10300	82400	42714	1,929
2000	13734	109872	56955	1,929
2500	17058	137368	71208	1,929
3000	20487	164928	85497	1,928
3500	23970	192448	99766	1,928
4000	27331	219928	114013	1,9298

BIBLIOGRAFIA

- Dispense a cura del docente
- Documentazione Java